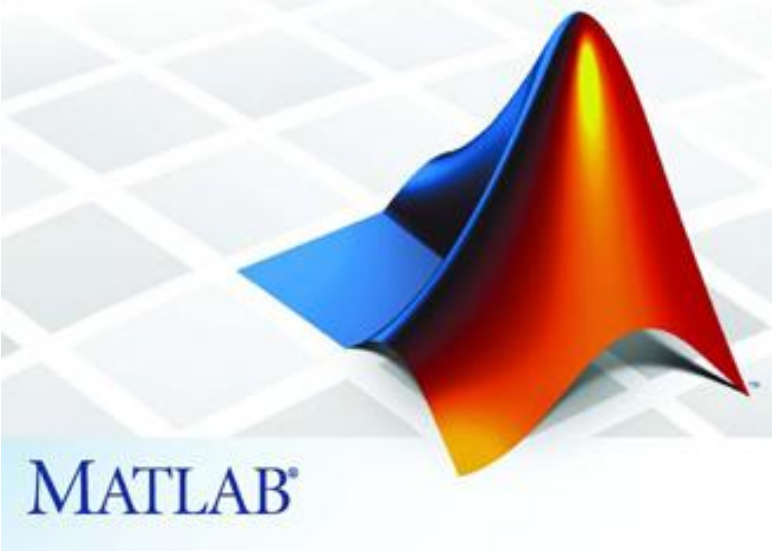
Sudan University of Science and Technology

School of Electronics Engineering

DSP LAB

LAB 01

Fourth year



INTRODUCTION

MATLAB (**mat**rix **lab**oratory) is a numerical computing environment and fourth-generation programming language. Developed by MathWorks, MATLAB allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages, including C, C++, Java, and Fortran.

It integrates *computation*, *visualization*, and *programming* environment.

Modes in Matlab:

1- Interactive.
2- Batch mode.

Advantages & features:

- Easy to use.
- Simple programming rules.
- Extendable.
- Linkage possibility.
- Transportability.

Matlab Applications:

- Control system.
- Signal processing.
- Image processing.
- Communications.
- Neutral network.
- Fuzzy logic.
- Others

Variables & data classes:

MATLAB variables are created with an assignment statement. The syntax of variable assignment is:

Variable name = a value (or an expression)

For example,

```
>> x = expression
```

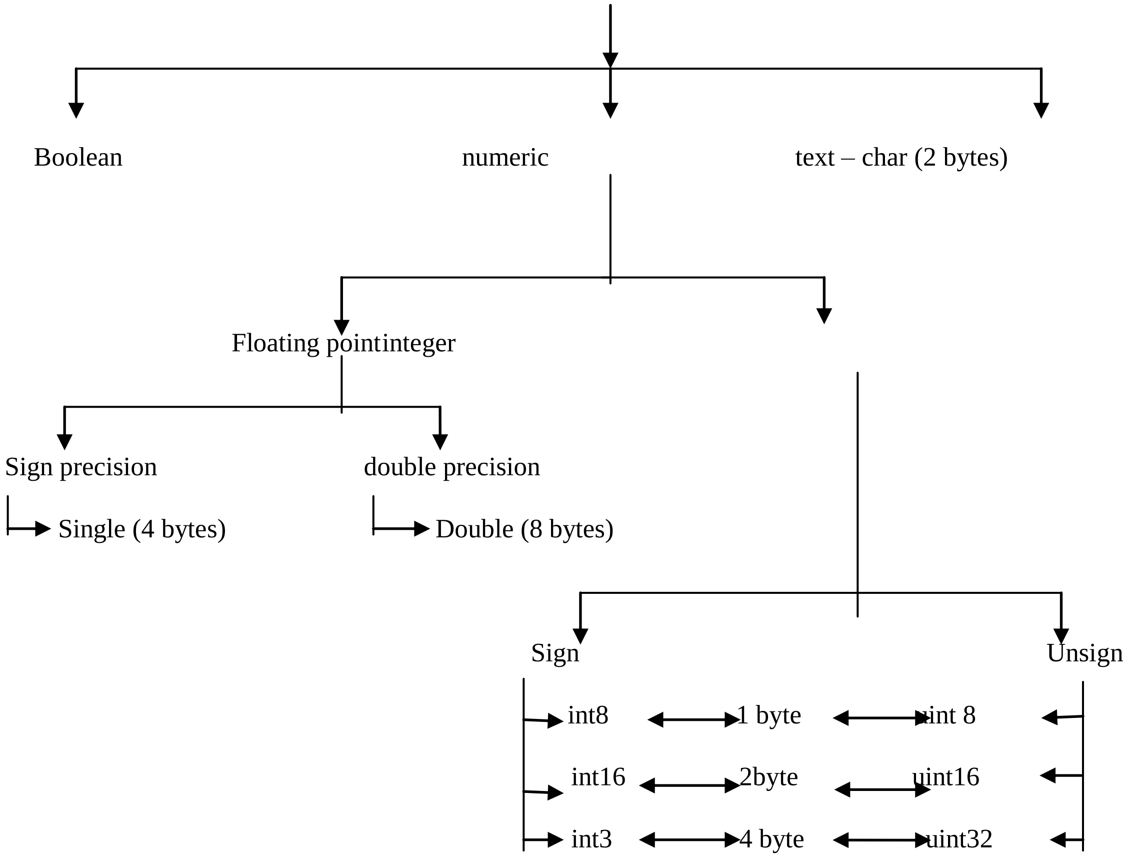
The expression is a combination of numerical values, mathematical operators, variables, and function calls. On other words, expression can involve:

- Manual entry.
- Built-in functions.
- User-defined functions.

Defining variable conditions:

1- No special characters ($, * , …).
2- No function name.
3- No reserved word (pi , I , j , inf , NAN , eps).
4- Case sensitive.

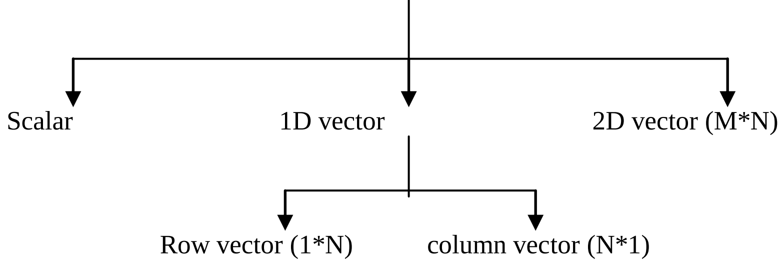
MATLAB classes



Double class is the default class for numbers.

Char class is default class for character

Basic data structure



Scalar :

e.g.: >> x = 90

1D vector :

- Row vector:

e.g.: >> x = [1 2 3 4 5 6]

- Column vector :

e.g.: >> x = $\begin{pmatrix}1\\2\\3\\4\\5\end{pmatrix}$

Functions:

- **Mathematical functions: (Examples)**

function	description	function	description
cos(x)	Cosine	abs(x)	Absolute value
sin(x)	Sine	sign(x)	Signum function
tan(x)	Tangent	max(x)	Maximum value
acos(x)	Arc cosine	min(x)	Minimum value
asin(x)	Arc sine	ceil(x)	Round towards positive infinity
atan(x)	Arc tangent	floor(x)	Round towards negative infinity
exp(x	Exponential	round(x)	Round to nearest integer
sqrt(x)	Square root	rem(x)	Remainder after division
log(x)	Natural logarithm	angle(x)	Phase angle
log10(x)	Common logarithm	conj(x)	Complex conjugate

- **Who& whose functions:**

These functions return or display all variables in the workspace of that function, and in the workspaces of all functions in which that function is nested.

- ***who***
  Show all used variables

Syntax:

```
>> who
```

Example: Show variable names starting with the letter `a`:

```
who a*
```

Example: Show variables stored in MAT-file `mydata.mat`:

```
who -file mydata
```

- ***whos***

Syntax:

```
>> whos
```

When the function is run, `whos` displays these attributes:

```
Name       Size          Bytes  Class     Attributes



c          1x3             48   double     complex
g          1x1              8   double     global
```

- ## Matrix generators

function	description
eye(m,n)	Returns an m-by-n matrix with 1 on the main diagonal
eye(n)	Returns an n-by-n square identity matrix
zeros(m,n)	Returns an m-by-n matrix of zeros
ones(m,n)	Returns an m-by-n matrix of ones
diag(A)	Extracts the diagonal of matrix A
rand(m,n)	Returns an m-by-n matrix of random numbers

- ## Linear spacing:

There is a command to generate linearly spaced vectors: *linspace*. It is similar to the colon operator (:), but gives direct control over the number of points.
For example,

```
>>y = linspace(a,b)
```

It generates a row vector y of 100 points linearly spaced between and including a and b.

```
>>y = linspace(a,b,n)
```

This is useful when we want to divide an interval into a number of subintervals of the same length.

• Output Functions:

Disp:

Displays text or array, `disp(X)` displays the contents of x variable, if `X` contains a text string, the string is displayed.

Another way to display an array on the screen is to type its name, but this prints a leading `"X="` which is not always desirable.

Note that *`disp`* does not display empty arrays.

Syntax:

```
disp(X)
```

• Input Functions:

Request user input , for example the line : *`evalResponse`* `= input(`*`prompt`*`)` displays the *`prompt`* string on the screen, waits for input from the keyboard, evaluates any expressions in the input, and returns the value in *`evalResponse`*. To evaluate expressions, the `input` function accesses variables in the current workspace

Syntax:

```
evalResponse = input(prompt)
strResponse = input(prompt, 's')
```

Matrix indexing:

Example:

$$A=\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

```
>> A(2,1)

ans =
      4
```

Correcting any entry is easy through indexing. Here we substitute A(3,3)=9 by A(3,3)=0. The result is

```
>> A(3,3) = 0
```

```
A =
          1  2  3
          4  5  6
          7  8  0
```

Colon operator in a matrix:

The colon operator can also be used to pick out a certain row or column. For example, the statement A(m:n,k:l specifies rows *m* to *n* and column *k* to *l*. Subscript expressions refer to portions of a matrix.

For example,

```
>> A(2,:)

          ans =
                    4   5   6
```

is the second row elements of A.

The colon operator can also be used to extract a sub-matrix from a matrix A.

```
>> A(:,2:3)
          ans =
                    2    3
                    5    6
                    8    0
```

A(:,2:3) is a sub-matrix with the last two columns of A.

A row or a column of a matrix can be deleted by setting it to a *null* vector, [ ].

```
>> A(:,2)=[ ]
          ans =
                    1   3
                    4   6
                    7   0
```

Creating a sub-matrix:

To extract a *submatrix* B consisting of rows 2 and 3 and columns 1 and 2 of the matrix A, do the following

```
>> B = A([2 3],[1 2])

          B =
                    4   5
                    7   8
```

To interchange rows 1 and 2 of A, use the vector of row indices together with the colon operator.

```
>> C = A([2 1 3],:)
```

```
C =
     4   5   6
     1   2   3
     7   8   0
```

It is important to note that the *colon operator* (:) stands for *all columns* or *all rows*. To create a vector version of matrix A, do the following

```
>> A(:)

     ans =
          1
          2
          3
          4
          5
          6
          7
          8
          0
```

The submatrix comprising the intersection of rows p to q and columns r to s is denoted by A(p:q,r:s).

As a special case, a colon (:) as the row or column specifer covers all entries in that row or column; thus

- A(:,j) is the jth column of A, while
- A(i,:) is the ith row, and
- A(end,:) picks out the last row of A.

The keyword end, used in A(end,:), denotes the last index in the specifed dimension. Here are some examples.

```
   >> A

     A =
          1   2   3
          4   5   6
          7   8   9

>> A (2:3,2:3)

          ans =
               5   6
               8   9

>> A (end:-1:1,end)
```

```
          ans =
               9
               6
               3

>> A([1 3],[2 3])

     ans =
          2   3
          8   9
```

Deleting row or column:

To delete a row or column of a matrix, use the *empty vector* operator, [ ].

```
>> A(3,:) = [ ]

     A =
          1   2   3
          4   5   6
```

Third row of matrix A is now deleted. To restore the third row, we use a technique for creating a matrix

```
>> A = [A(1,:);A(2,:);[7 8 0]]

     A =
          1   2   3
          4   5   6
          7   8   0
```

Matrix A is now restored to its original form.

Dimension:

To determine the *dimensions* of a matrix or vector, use the command size. For example,

```
>> size(A)

     ans =
          3   3
```

means 3 rows and 3 columns.

Or more explicitly with,

```
>> [m,n]=size(A)
```

Transposing a matrix:

```
>> A'
        ans =
                1   4   7
                2   5   8
                3   6   0
```

Representing Polynomials

MATLAB represents polynomials as row vectors containing coefficients ordered by descending powers. For example, consider the equation

$$p(x) = x^3 - 2x - 5$$

To enter this polynomial into MATLAB, use

```
» p = [1 0 -2 -5];
```

Polynomial Roots

The roots function calculates the roots of a polynomial.

```
» r = roots (p)
r =
2.0946
-1.0473 + 1.1359i
-1.0473 - 1.1359i
```

By convention, MATLAB stores roots in column vectors. The function poly returns to the polynomial coefficients.

```
» p2 = poly(r)
p2 =
1 8.8818e-16 -2 –5
```

So the poly and roots are inverse functions.

Polynomial Evaluation

The polyval function evaluates a polynomial at a specified value. To evaluate p at *s* = 5, use

```
» polyval(p,5)
        ans =
        110
```

It is also possible to evaluate a polynomial in a matrix sense. In this case $p(x) = x^3 - 2x - 5$ becomes $p(X) = X^3 - 2X - 5I$, where *X* is a square matrix and *I* is the identity matrix. For example, create a square matrix X and evaluate the polynomial p at X.

```
»X = [2 4 5; -1 0 3; 7 1 5];
»Y = polyvalm(p,X)
Y =
377 179 439
111 81 136
490 253 639
```

Flow Control

MATLAB has four flow control structures: the if statement, the for loop, the while loop, and the switch statement.

1- If Statement

The simplest form of the if statement is

```
if         expression
           statements
end
```

The statements are executed if the elements of expression are all nonzero. For example, the following code swaps x and y if x is greater than y:

```
if x > y
        temp = y;
        y = x;
        x = temp;
end
```

When an if statement is followed on its line by further statements, a comma is needed to separate the if from the next statement:

```
if x > 0, x = sqrt(x); end
```

Statements to be executed only if expression is false can be placed after else, as in the example

```
e = exp(1);
if 2^e > e^2
        disp('2^e is bigger')
else
        disp('e^2 is bigger')
end
```

Finally, one or more further tests can be added with elseif (note that there must be no space between else and if):

```
if isnan(x)
        disp('Not a Number')
elseif isinf(x)
        disp('Plus or minus infinity')
else
        disp('A ''regular'' floating point number')
end
```

2- For Loop

The for loop is one of the most useful MATLAB constructs although, , experienced programmers who are concerned with producing compact and fast code try to avoid for loops wherever possible. The syntax is

```
for variable =    expression
                  statements
end
```

Usually, expression is a vector of the form i:s:j (see Section 5.2). The statements are executed with variable equal to each element of expression in turn. For example, the sum of the first 25 terms of the harmonic series 1/i is computed by :

```
>> s = 0;
>> for i = 1:25, s = s + 1/i; end, s
s =
3.8160
```

Another way to define expression is using the square bracket notation:

```
for x = [pi/6 pi/4 pi/3], disp([x, sin(x)]), end

0.5236 0.5000
0.7854 0.7071
1.0472 0.8660
```

Multiple for loops can of course be nested, in which case indentation helps to improve the readability. The following code forms the 5-by-5 symmetric matrix A with (i, j) element i/j for j ≥ i:

```
n = 5; A = eye(n);
for j = 2:n
        for i = 1:j-1
                A(i,j) = i/j;
                A(j,i) = i/j;
        end
end
```

3- While Loop

The while loop has the form

```
while    expression
         statements
end
```

The statements are executed as long as expression is true. The following example approximates the smallest nonzero floating point number:

```
>> x = 1; while x>0, xmin = x; x = x/2; end, xmin
        xmin =
        4.9407e-324
```

A while loop can be terminated with the break statement, which passes control to the first statement after the corresponding end. An infinite loop can be constructed using while 1, ..., end, which is useful when it is not convenient to put the exit test at the top of the loop. (Note that, unlike some other languages, MATLAB does not have a “repeat-until” loop.) We can rewrite the previous example less concisely as

```
x = 1;
while 1
        xmin = x;
        x = x/2;
        if x == 0, break, end
end
xmin
```

The break statement can also be used to exit a for loop. In a nested loop a break exits to the loop at the next higher level. The continue statement causes execution of a for or while loop to pass immediately to the next iteration of the loop, skipping the remaining statements in the loop.

As a trivial example :

```
for i=1:10
if i < 5, continue, end
disp(i)
end
```

The output displays the integers 5 to 10. In more complicated loops the continue statement can be useful to avoid long-bodied if statements.

4- Switch Statement

The final control structure is the switch statement. It consists of “switch expression” followed by a list of “case expression statements”, optionally ending with “otherwise statements” and followed by end. The switch expression is evaluated and the statements following the first matching case expression are executed. If none of the cases produces a match then the statements following otherwise are executed. The next example evaluates the p-norm of a vector x (i.e., norm(x,p)) for just three values of p:

```
switch p
        case 1
                y = sum(abs(x));
        case 2
                y = sqrt(x’*x);
        case inf
                y = max(abs(x));
        otherwise
                error(’p must be 1, 2 or inf.’)
end
```

The expression following case can be a list of values enclosed in parentheses. The switch expression then matches any value in the list:

```
x = input(’Enter a real number: ’);
switch x
        case {inf,-inf}
```

```
                        disp('Plus or minus infinity')
                case 0
                        disp('Zero')
                otherwise
                        disp('Nonzero and finite')
        end
```

[1]

• Basic plotting

Overview

Plotting a given data set or the results of computation is possible with very few commands.

- Creating simple plots (*Plot* function)

The basic MATLAB graphing procedure, for example in 2D, is to take a vector of *x*- coordinates, x = ($x_1; : : : ; x_N$), and a vector of *y*-coordinates, y = ($y_1; : : : ; y_N$), locate the Points ($x_i; y_i$), with $i = 1; 2; : : : ; n$ and then join them by straight lines. You need to prepare *x* and *y* in an identical array form; namely, *x* and *y* are both row arrays or column arrays of the *same* length.

The MATLAB command to plot a graph is plot(x,y). The vectors x = ($1; 2; 3; 4; 5; 6$)
and y = ($3;¡1; 2; 4; 5; 1$) produce the picture shown in Figure 1.1.

```
>> x = [1 2 3 4 5 6];
>> y = [3 -1 2 4 5 1];
>> plot(x,y)
```

Note: The plot function has different forms depending on the input arguments. If y is a vector plot(y) produces a piecewise linear graph of the elements of y versus the index of the elements of y. If we specify two vectors, as mentioned above, plot(x,y) produces a graph of y versus x.

---

[1] Reference: Copyright ©2005 by the Society for Industrial and Applied Mathematics

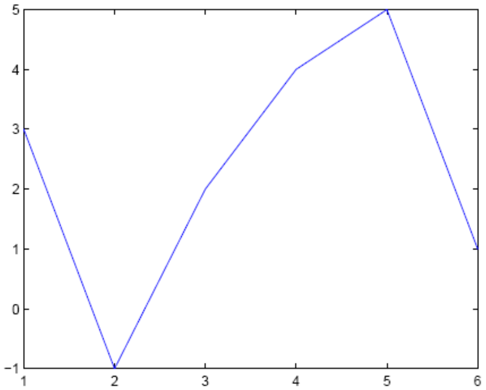


Figure 1.1: plotting x vs. y

- Multiple data sets in one plot

Multiple (*x; y*) *pairs* arguments create *multiple* graphs with a single call to *plot*. For example, these statements plot three related functions of *x*: *y*1 = 2 cos(*x*), *y*2 = cos(*x*), and *y*3 = 0.5 cos(*x*), in the interval 0 ≤ *x* ≤ 2¼.

```
>> x = 0:pi/ 100:2*pi;
>> y1 = 2*cos(x);
>> y2 = cos(x);
>> y3 = 0.5*cos(x);
>> plot(x,y1,'--',x,y2,'-',x,y3,':')
>> xlabel('0 \leq x \leq 2\pi')
>> ylabel('Cosine functions')
>> legend('2*cos(x)','cos(x)','0.5*cos(x)')
>> title('Typical example of multiple plots')
>> axis([0 2*pi -3 3])
```

The result of multiple data sets in one graph plot is shown in Figure 1.2.

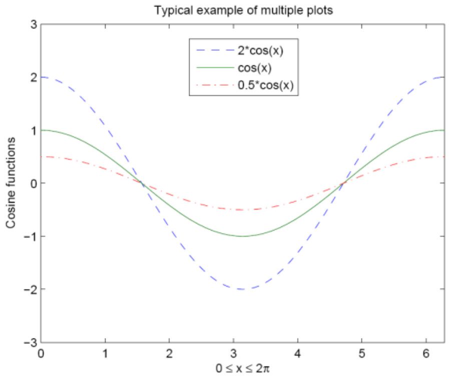


Figure 1.2: Typical Example of multiple plots.[2]

For More details on plotting:
http://ocw.usu.edu/Civil_and_Environmental_Engineering/Numerical_Methods_in_Civil_Engineering/MatlabElementaryGraphs.pdf

---

[2] Introduction To Matlab For Engineering Students David Houcque Northwestern University

- Creating simple plots (*Stem* function)

Syntax

- `stem(Y)`
- `stem(X,Y)`

Two-dimensional stem plot displays data as lines extending from a baseline along the *x*-axis. A circle (the default) or other marker whose *y*-position represents the data value terminates each stem.

- `stem(Y)` plots the data sequence `Y` as stems that extend from equally spaced and automatically generated values along the *x*-axis. When `Y` is a matrix, `stem` plots all elements in a row against the same *x* value.
- `stem(X,Y)` plots `X` versus the columns of `Y`. `X` and `Y` are vectors or matrices of the same size. Additionally, `X` can be a row or a column vector and `Y` a matrix with `length(X)` rows.

Example:

Create a stem plot of a circular function.

- `y = linspace(0,2*pi,10);`
- `h = stem(cos(y),'fill','-.');`
- `set(h(3),'Color','r','LineWidth',2) % Set base line properties`
- `axis ([0 11 -1 1])`

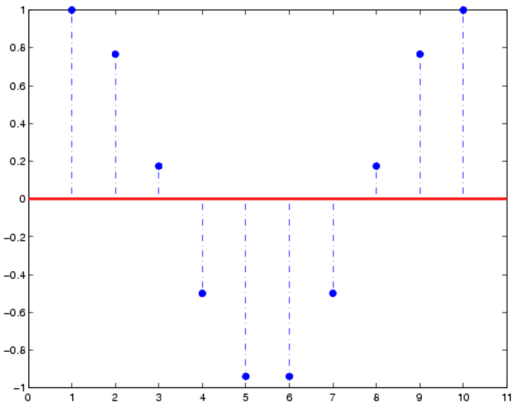


Figure 1.3 : Resulting graph.

Plots to an Existing Graph
The hold command enables you to add plots to an existing graph. When you type

```
» hold on
```

MATLAB does not replace the existing graph when you issue another plotting command; it adds the new data to the current graph, rescaling the axes if necessary. To remove the effect of the **hold on** function you may use:

```
» hold off
```

Multiple Plots in One Figure

The subplot command enables you to display multiple plots in the same window or print them on the same piece of paper. Typing:

```
» subplot(m,n,p)
```

partitions the figure window into an m-by-n matrix of small subplots and selects the pth subplot for the current plot. The plots are numbered along first the top row of the figure window, then the second row, and so on. For example, these statements plot data in four different subregions of the figure window.

```
t = 0:pi/10:2*pi;
[X,Y,Z] = cylinder(4*cos(t));
subplot(2,2,1); mesh(X)
subplot(2,2,2); mesh(Y)
subplot(2,2,3); mesh(Z)
subplot(2,2,4); mesh(X,Y,Z)
```

• m-files

The problem in the commands entered in the Command Window is that thew commands cannot be saved and executed again for several times. Therefore, a different way of executing repeatedly commands with MATLAB is:

1. to *create* a file with a list of commands,
2. *save* the file, and
3. *run* the file.

If needed, corrections or changes can be made to the commands in the file. The files that are used for this purpose are called script files or *scripts* for short.

This section covers the following topics:

- M-File Scripts
- M-File Functions

a- M-File Scripts

A *script file* is an external file that contains a sequence of MATLAB statements. Script files have a file name extension .m and are often called M-files. M-files can be *scripts* that simply execute a series of MATLAB statements, or they can be *functions* that can accept arguments and can produce one or more outputs.

Examples

Here are two simple scripts.

Example 1

Consider the system of equations:

$$x + 2y + 3z = 1$$
$$3x + 3y + 4z = 1$$
$$2x + 3y + 3z = 2$$

Find the solution x to the system of equations.

Solution:

- Use the MATLAB *editor* to create a file: File → New → M-file.
- Enter the following statements in the file:

```
A = [1 2 3; 3 3 4; 2 3 3];
b = [1; 1; 2];
x = A\b
```

- Save the file, for example, example1.m.
- Run the file, in the command line, by typing:

```
>> Example1
          x =
                    -0.5000
                    1.5000
                    -0.5000
```

When execution completes, the variables (A, b, and x) remain in the workspace. To see a listing of them, enter whos at the command prompt.

Note: The MATLAB editor is both a text editor specialized for creating M-files and a graphical MATLAB debugger. The MATLAB editor has numerous menus for tasks such as *saving*, *viewing*, and *debugging*. Because it performs some simple checks and also uses colour to differentiate between various elements of codes, this text editor is recommended as the tool of choice for writing and editing M-files.

There is another way to open the editor:

```
>> edit
```

or

```
>> edit filename.m
```

To open filename.m. in the Command Window.

Script side-effects

All variables created in a script file are added to the workspace. This may have undesirable effects, because:

- Variables already existing in the workspace may be overwritten.
- The execution of the script can be affected by the state variables in the workspace.

As a result, because scripts have some undesirable side-effects, it is better to code any complicated applications using rather function M-file.

b- M-File functions

Functions are programs (or *routines*) that accept *input* arguments and return *output* arguments. Each M-file function (or *function* or *M-file* for short) has its *own* area of workspace, separated from the MATLAB base workspace.

This simple function shows the basic parts of an M-file.

```
function f = factorial(n)
% FACTORIAL(N) returns the factorial of N.
% Compute a factorial value.
f = prod(1:n);
```

The first line (1) of a function M-file starts with the keyword function. It gives the function *name* and order of *arguments*. In the case of function factorial, there are up to one output argument and one input argument.
As an example, for *n* = 5, the result is,

```
>> f = factorial(5)
         f =
         120
```

In addition, it is important to note that *function name* must begin with a letter, and must be no longer than the maximum of 63 characters. Furthermore, the name of the text file that you save will consist of the function name with the extension .m. Thus, the above example file would be factorial.m.

Table 1.1 summarizes the differences between *scripts* and *functions*.

Scripts	Functions
Do not accept input arguments or return output arguments.	Can accept input and return output arguments.
Store variables in a workspace that is shared with other scripts	Store variables in a workspace internal to the function.
Are useful for automating a series of commands	Are useful for extending the MATLAB language for your application

[3]

---

[3] Reference : Introduction To Matlab For Engineering Students, David Houcque Northwestern University

Exercises:

1) a. Generate a matrix of size 4x4 of normally distributed random numbers.
   b. Change the value of the 3rd column to [6 9 2 5].
   c. Delete the 2nd row.

2) Consider the polynomial $p(x) = x^4 - 5x^3 + 7x - 10$
   1. Find the roots of this polynomial.
   2. From these roots, reconstruct p(x).
   3. Find the value of p(x) at x=3.
   4. Evaluate the polynomial at X (matrix sense).

3) Using MATLAB, plot the function

$$f(t) = e^{-2t} \cos(3 \pi t)$$

Where t ranges from –2 to 10. Give the x-axis the title (t sec) and the y-axis the title (f(t) volts). Repeat using stem function. Then use subplot to combine the two figures in one figure and give it the name: “difference between plot and stem”.